Code Generation From Hierarchical Concurrency Specifications

Denis Roegel, Scott Smolka

Code Generation From Hierarchical Concurrency Specifications*

(October 8, 2001 version)

Denis Roegel[1] and Scott Smolka[2]

[1] LORIA, Campus scientifique, BP 239
54506 Vandœuvre-lès-Nancy, France
roegel@loria.fr,
WWW home page: http://www.loria.fr/~roegel
[2] Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400, USA
sas@cs.sunysb.edu,
WWW home page: http://www.cs.sunysb.edu/~sas

Abstract. This paper explains how executable Java code is generated from hierarchical specifications in the Concurrency Factory specification and verification environment. Besides the ability to generate executable code from verified, abstract concurrency specifications, the paper's main contributions include: (1) new solutions to the well-known input/output guard-scheduling problem in the context of hierarchically configured concurrent systems; (2) code-generation algorithms that produce both thread-based Java code and distributed Ada '95 code; (3) the use of the Concurrency Factory itself to verify an abstraction of each generated code module; in this sense, the Factory is self-verifying; and, finally, (4) a report on our experience in executing the generated code for simulation and debugging purposes in the case of the Rether real-time ethernet protocol.

1 Introduction

The Concurrency Factory [CLSS96a,CLSS96b] is an integrated toolset for the specification and verification of concurrent systems. The main features of the Factory are an interactive editor and simulator of hierarchical concurrency specifications, a local model checker with partial-order reduction for the modal mu-calculus [RS97], and an equivalence checker based on Milner's observational equivalence. A previous release of the Factory also provided a specifications compiler that produced executable Facile code from Factory concurrency specifications. Facile [TLK96] is a distributed programming language based on a combination of the functional language ML, Milner's CCS, and Hoare's CSP.

* Research supported in part by AFOSR grant F49620-95-1-0508 and NSF grant CCR–9505562.

The presence of a specifications compiler in a system like the Concurrency Factory is important: such specifications can be simulated in the Factory but not executed independently of the Factory. In essence, a specifications compiler relieves the user of the burden of manually recoding their designs in the target language of their final system.

As in CCS, Facile allows input and output commands to appear as guards in nondeterministic choice statements, and this greatly simplified the process of translating Concurrency Factory specifications into the language. However, the Facile implementation of nondeterministic choice is deterministic, with the first enabled alternative selected for execution. Moreover, Facile has not yet gained widespread acceptance in the systems-implementation arena.

To address these concerns, we present a new code-generation facility for the Concurrency Factory. The user can direct the Factory to produce either `Java` or Ada '95 code, both of which can be considered mainstream systems-programming languages. Moreover, the code produced *faithfully* preserves the semantics of nondeterministic choice.

The main contributions of this paper can be seen as the following:

- We describe in detail new algorithms for generating executable, stand-alone code from verified, hierarchical, abstract concurrency specifications. The resulting code can be viewed as a rapid prototype of the target system in which the communication and synchronization aspects of the system are fully realized and in which "placeholders" are provided for the functional aspects of the system not fully captured in the original specification.
- The code-generation algorithms and their implementations in the Concurrency Factory constitute new solutions to the well-known input/output guard-scheduling problem [BS83,FR80] in the context of hierarchically configured concurrent systems.
- The Factory can produce either thread-based `Java` code or distributed Ada '95 code. We thus address the code-generation problem for both parallel, shared-memory multiprocessor architectures, and for distributed platforms supporting remote procedure call.
- The Factory produces a VPL abstraction of the generated code each time the code-generation module is invoked. Using the Factory's model checker, the VPL specification can then be checked against a standardized suite of modal mu-calculus formulas for the purpose of ensuring the correctness of the generated code. In this sense, the Factory is self-verifying.
- By executing the generated code—possibly enriched with additional application-dependent functional code—on the target execution platform, a means for simulating and debugging the rapid prototype produced by the Factory is attained. We report on our experience in this regard in the context of the Rether real-time Ethernet protocol for multimedia applications [VC95].

In terms of related work, a number of specification and verification tool suites provide some form of code generation, differing in terms of the speci-

fication language supported and the language of the generated code. Examples includes Statemate [HLN+90], generating C/Ada from Statecharts, SystemSpecs [Ivy96], generating C/C++/occam/VHDL from high-level Petri Nets, Open Cæsar [Gar98], generating C from LOTOS specifications, and the IOA tool [GL98], generating `Java` from I/O Automata. To our knowledge, none of these systems generates code that faithfully mirrors the semantics of input/output guards in the context of hierarchical concurrency specifications.

The structure of the rest of the paper is as follows. Section 2 describes the specification languages of the Factory. Our scheduling algorithms for input/output guards are the topic of Section 3 while Section 4 focuses on the `Java` implementations of the algorithms. Section 5 discusses the use of the Factory's local model checker to verify the correctness of the generated code while Section 6 reports on our experience in simulating the generated code for debugging purposes. Some concluding remarks are offered in Section 7.

Because of space considerations, we focus our attention in this paper on the translation of Concurrency Factory specifications into `Java` parallel threads [Fla97,OW97]. The generation of RPC-based Ada '95 code is described in detail in [Ngu98] and is performed in a manner similar to the `Java` code generation.

2 Specification Languages of the Factory

Concurrency specifications in the Concurrency Factory are either graphical, textual, or hybrid, that is, a combination of both. The graphical specification language of the Factory is GCCS (Graphical CCS) and the textual language is VPL (Value-Passing Language). Specifications are organized hierarchically into network nodes and process nodes, with network nodes as the internal nodes and process nodes at the leaves. A typical Concurrency Factory hierarchical specification is illustrated in Figure 1. The layout of a VPL specification is statical and cannot change over time.

VPL subsumes GCCS in expressive power and we will therefore use it to explain how code generation in the Factory is performed. As indicated above, a specification in VPL is a tree-like hierarchy of systems. A system is either a network or a process. A network consists of a collection of subsystems running in parallel and communicating with each other over typed channels.

Channel names are made known to a system through explicit channel declarations or via parameter passing. That is, each system definition is parameterized by a list of channels, and the system in question may refer to channels supplied to it as actual parameters when the system is instantiated. Channels of the former kind are said to be *local* to the system, and those of the latter kind are said to be *non-local*.

VPL-supported data types include integers of bounded size and arrays and records composed of such integers. A VPL process is a sequence of statements. Simple statements of VPL are assignments of arithmetic or boolean expressions to variables, input/output operations on channels, and the `skip` statement. Input

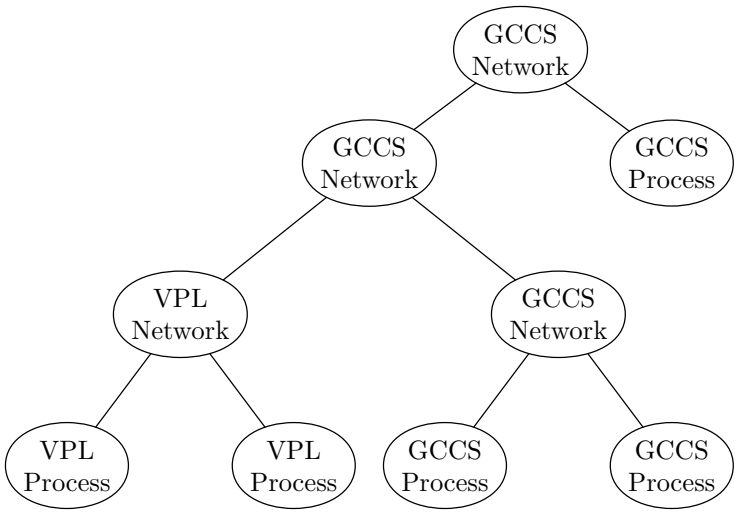

Fig. 1. Example hierarchical specification.

statements are of the form `a?v` where `a` is a channel and `v` is a variable. Output statements take the form `a!e` where `a` is a channel and `e` is an expression. As in CCS, communication occurs in VPL when *complementary* input and output statements are executed. The transmission of dataless signals over a channel `a` is achieved by writing `a?*` and `a!*`. Channel `a` in this case is said to be of type `synch`.

Complex statements include sequential composition, `if-then-else`, `while-do`, and nondeterministic choice in the form of the `select` statement. `Select` alternatives are separated by a percent sign `%`. VPL also supports a mechanism for declaring procedures. Comments in VPL are delimited by curly braces `{` and `}`. A very simple VPL system is shown in figure 2.

The semantics of the `select` statement is sketched in figure 3.

The operational semantics of VPL is given in [Tiw97]. The enabled transitions are defined as in value-passing CCS: communication and synchronization occur between two processes, resulting in the internal action τ, and local channels in a network correspond to restricted pairs of complementary ports in CCS.

3 Scheduling Algorithm for Input/Output Guards in Hierarchical Systems

A faithful generation of code from VPL specifications requires that the input/output guard scheduling problem [BS83,FR80] be solved. The solutions we propose are new variants of previously published solutions and are targeted toward hierarchically configured systems of concurrent processes.

```
type int 10
network n1()
  begin
    process p1(a: synch,b: synch)
      begin
        var x: int
        x:=0;
        while true do
          select
            a!*
            %
            b?*
            %
            if (x=0) a! else x:=x+1
          end
        end
      end; {end p1}
    process p2(a: synch,b: synch)
      begin
        while true do
          select
            b!*
            %
            a?*
            %
            a!*
          end
        end
      end; {end p2}
    channel a: synch
    channel b: synch
    p1(a,b)|p2(a,b)
  end; {end n1}
```

Fig. 2. Example of a simple VPL program

$$\frac{a!e_1|a?e_2 \longrightarrow^{\tau} e_1|e_2}{(a!e_1 + f_1)|(a?e_2 + f_2) \longrightarrow^{\tau} e_1|e_2}$$

$$\frac{a!e_1|o.e_2 \longrightarrow^{o} a!e_1|e_2}{a!e_1|(o.e_2 + f) \longrightarrow^{o} a!e_1|e_2}$$

where o is a noncommunicating operation (such as `skip`).

Fig. 3. Semantics of the `select` statement (nondeterministic choice)

The guard-scheduling problem has been elegantly characterized by Chandy and Misra as one of *committee coordination* [CM88]: each professor in a university serves on one or more committees, the members of which are fixed; furthermore, a committee cannot convene until all its members are present. The problem then is to schedule committee meetings such that no two committees with a common member convene simultaneously. Since we are considering CCS-style communications, each involving two processes, the committees of interest in this paper are of size two.

It is well known that allowing processes to choose independently from among their communication alternatives will not lead to a correct solution to the guard-scheduling problem. Such an approach is inherently prone to deadlock. Rather, one must resort to symmetry breaking, randomization, or centralized scheduling. Our approach is a hybrid centralized/distributed one. Subsystems communicate their alternatives at runtime (since they can depend on guards) to their parent networks, and a parent attempts to schedule a communication each time it receives new input from one of its children. If some of the channels involved are non-local to the parent network, the communication alternatives the parent network knows about are sent to its parent, and so on. Eventually, this propagation will stop and if some communication can be scheduled, it is propagated back down to the appropriate processes.

We examine three algorithms. The first one uses locks to prevent race conditions and each network sends all its alternatives to its parent. The second one is a simplification where only incoming alternatives (as opposed to all alternatives pending in a network) are carried through and no locks are used. This algorithm remains simple as long as a specific condition is met. The third algorithm doesn't use locks and tries to preserve locality. Messages can go both ways (up and down) before being handled. Moreover, this third algorithm is probabilistic.

3.1 Algorithm using locks

This algorithm relies heavily on locks in order to avoid a message crossing (see figure 7).

We present the algorithm in VPL-style pseudo-code, and in two parts: the code for a generic process and the code for a generic network. The pseudo-code for a process is given in figure 4.

A process alternates between local and non-local code. Non-local code is either a `select` statement or an input/output command. The alternatives (which can consist of communications or noncommunicating statements such as guards) are communicated to the parent network, after which the process waits for an answer. When an answer arrives, the process performs the scheduled operation.

The pseudo-code for a network is given in Figure 5.

A network either (i) receives an answer from its parent, (ii) waits for an acknowledgement from its parent, or (iii) tries to input and process data from a child (only when the network is not waiting for an acknowledgment). An acknowledgement means that the parent has finished processing the set of alternatives the network sent it previously, and the parent lock can safely be returned to

while *true*
  <local code>
  send *choices* $a_1, \ldots, a_k$ *to parent*
  receive *answer* a_i *from parent*
  perform *the answer* a_i
end while

Fig. 4. Pseudo-code for a process (algorithm 1)

the parent. The details of the algorithm's use of locks are discussed below. The presence of `skip` alternatives in Figures 5 and 6 are to allow a network to proceed asynchronously with respect to its parent or children. If a network is the root network and therefore has no parent, the pseudo-code of Figure 5 reduces to processing messages from its children.

The pseudo-code a network executes to process messages from its children, case (iii) above, is given in Figure 6. The network first passes the lock to one of its children after which it may receive a non-empty list of communication alternatives from that child. The communications will be either pairs of messages (such as `a!` and `a?`) or messages alone (such as `c!`). The network then waits for its parent to send its lock or to send an answer (to a message the network previously sent its parent). In the latter case, the answer is interpreted and the answer (which communications or operations have been scheduled) is forwarded to the appropriate children. In either case, the network now has the lock and this means it can talk freely with its parent (see below). The network now computes new combinations of complementary communication alternatives and new operations to perform (such as skips) using the data it received from the child. If there are no candidate operations (for instance, if the network has the choice between `a!` and `b!`, but with `a` and `b` being both local channels to the network), the lock is returned to the parent. If there are one or more candidate operations, then two cases arise: (i) There is at least one operation that is not local to the network, in which case all operations are forwarded to the parent. The lock is not returned and the network sets *parentAck* to false to indicate that an acknowledgement is still pending. (ii) All operations are local, in which case the network chooses one and interprets it. The lock is returned since the network is not waiting for an acknowledgement from its parent. The operation chosen is either a single operation involving no communication (this operation occurred in a `select` statement) or a communication between two processes (and, possibly, but non compulsorily, two children of the network).

The pseudo-code of Figure 6 must be modified slightly in the case of the root network of the system. In that case, send and receive operations involving the network's parent network (of which there is none) should be replaced by `skip` statements and the code simplified accordingly.

```
parentAck ← true // we are not waiting for the parent
while true
   select
      % // alternative 1
      begin
         receive answer from parent
         interpret answer and forward to appropriate children
      end
      % // alternative 2
      begin
         if (parentAck=false)
            then
               select
                  %
                  begin
                     receive acknowledgment from parent
                     parentAck ← true
                     return parent lock to parent
                  end
                  %
                  skip // no acknowledgment received
               end select
            else
               process messages from children (see Figure 6)
         end if
      end
   end select
end while
```

Fig. 5. Pseudo-code for a network (main part) (algorithm 1)

Locks are used to avoid race conditions. In particular, messages going upwards in the hierarchy can cross messages going downwards, leading to deadlock. An example is given in figure 7. The scenario is the following:

Process P_1 wants to perform an $a!*$ and sends it to N_1; network N_1 forwards this communication request to N_3. Process P_3 wants to perform an $a?*$ and sends it to N_2; network N_2 forwards it to N_3. Network N_3 decides to pair $a!*$ of P_1 and the $a?*$ of P_3. Process P_2 wants to perform an $a?*$. The following two events can now occur in the absence of locks: (i) N_1 sends $(a?*, a!*, \tau)$ to N_3; (ii) N_3 sends $a!*$ back to N_1. Events (i) and (ii) represent a crossing of two messages that can lead to deadlock as N_3 is allowed to choose among alternatives that are no longer possible.

In the correct implementation crossings are prevented through the use of locks. All children of a network share a lock (the "parent lock") to access the

```
select
   begin
      send lock to a child network
      select
         receive lock back from the child
         %
         begin
            receive list of alternatives from the child
            select
               receive lock from parent
               %
               begin
                  receive answer from parent
                  interpret the answer and forward messages to children if necessary
                  receive lock from parent
               end
            end select
            compute combinations
            if (number of combinations > 0)
               then
                  if (number of non-local communications > 0)
                     begin
                        send combinations to parent
                          // (including the local ones, as τ)
                        parentAck ← false
                     end
                  else
                     begin
                        choose a combination // for instance randomly
                        interpret the combination and forward messages if necessary
                        // (send it to the child or split it and send it to the children)
                        return parent lock to parent
                     end
                  end if
               else
                  return parent lock to parent
            end if
            send acknowledgement to child that sent alternatives
               (if the child is a network)
            receive lock back from child (if the child is a network)
         end
      end select
   end
   %
   skip // the child does not ask for the lock
end select
```

Fig. 6. Pseudo-code for processing messages from children (algorithm 1)

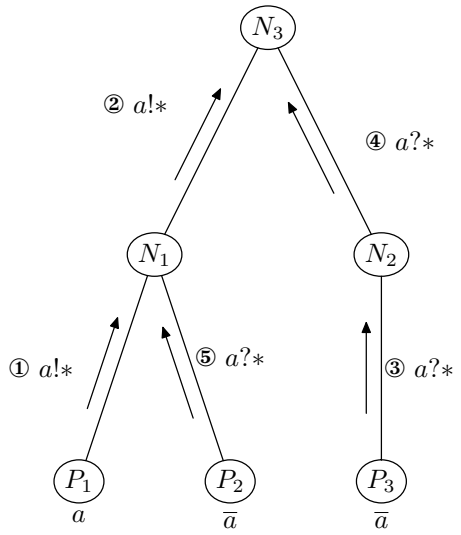

Fig. 7. Crossing of messages.

parent. Once a child has the lock, the parent cannot be called by another child. The lock is kept until the parent finishes analyzing the messages sent by the child, or until the parent provides an answer. Only then can another child get the parent lock, and compute combinations to send to the parent. In this way, if a parent has delivered an answer to a child, the child is guaranteed to have taken the answer into account before sending a subsequent message to the parent.

3.2 Algorithms without locks

In these two algorithms, only the new alternatives are sent to the parents.

Algorithm 2 It is possible to simplify the scheduling algorithm through the removal of locks, but for this, one special case has to be dispensed: if a process has an alternative, it must be between channels that are declared at the same level in the hierarchy. If this is not the case, and if we still want to keep locality (that is, if we want to avoid messages from being handled too much out of scope), we have to use the third algorithm. The algorithm for a network is given in figure 8. Processes are unchanged.

The algorithm assumes a different data structure than the one using locks. Each network keeps track of incoming messages. These messages come from a process, but the network gets the messages through a child. In the algorithm with locks, a network would usually receive not only "sends" and "receives," but also τs corresponding to matches down in the hirearchy (this being meant to avoid a bias towards local communications). The messages received by a network would correspond to all the pending messages of one of its children.

In the new algorithm, only a set of messages corresponding to a specific process would reach the network at a time. If the process only issues a send or a receive, the message will be made of just that. If the process has a choice (with `select`) all the alternatives will be sent. Later, a given message will be sent by a network at most once. This will avoid the problem of crossing messages.

Naturally, a network can store its pending messages by children, but it will have to keep track of the process from which they originate, in order to cancel the other alternatives, if one is chosen.

The algorithm without locks has one catch: it will be difficult to handle the case where a process has a choice between sending (for instance) through the channel a and receiving through the channel b, when a and b are not declared at the same level (and therefore at the same place). For it would mean that the alternative has to be split and one part of the messages goes higher in the hierarchy than the other one.

Algorithm 3 The problem of the previous algorithm can be dealt with as follows. This algorithm tries to keep locality as much as possible. Each process behaves like in the previous algorithm and sends its list of alternatives to its parent network.

```
while true
   select
      % // alternative 1
      begin
         receive answer from parent
         interpret answer and forward to appropriate children
      end
      % // alternative 2
      begin
         receive new list of alternatives from a child
         if the alternatives are all local
            if there is a local match
               forward the match to appropriate children
               and delete the information from this network
            else
               store the alternatives in the network
            end if
         elseif the alternatives are all non-local
            send the alternatives to the parent
         else
            // THIS CASE IS NOT ALLOWED
         end if
      end
      % // alternative 3
      skip
   end select
end while
```

Fig. 8. Pseudo-code for a network (algorithm 2)

In order to explain the behavior of a network, we first need to define a few notions:

- a message is basically a list of alternatives received by a network, plus some information to track its origin (list of processes and networks) and, for each channel involved in a communication, the level at which this channel is defined, if known; an example of message is $N_{n_i} N_{n_{i-1}} \dots N_{n_1} P_j(a^{(N_2)}!*, b!*)$;
- a channel is *sublocal* if it is declared by a descendant of the current network;
- a channel is *local* if it is declared by the current network;
- a channel is *superlocal* (or *non local*) if it is declared at a higher level;
- a message (or one of its alternatives, or a match) is *sublocal* if all channels involved are sublocal;
- a message (or one of its alternatives) is *mixed* or *mixed (sub)local/non local* if some, but not all, channels are non local;
- a message (or one of its alternatives, or a match) is *superlocal* (or *non local*) if all channels involved are non local.

A network does one of several things (pseudo-code is given in figures 9 and 10):

1. *it receives a message from a child*; this input is a list of alternatives as produced by a process, for instance $P_1(a!*, b!*)$; some of the alternatives may be non-communicating ones, such as skips; a network knows from which child the list originates; if this network (say N_1) hands this list over to its parent (say N_3), N_3 will mark the list as coming from N_1, in addition to the P_1 origin; so N_3 might manipulate something like $N_1P_1(a!*, b!*)$. Thus, the list of alternatives is prefixed by a list of nodes. In general, a network receives messages like $N_{n_i}N_{n_{i-1}} \dots N_{n_1}P_j(\langle\text{list of alternatives}\rangle)$. Moreover, some channels in the list may be *sublocal* and in this case the network where the channel is local is added as an index, for instance $N_{n_i}N_{n_{i-1}} \dots N_{n_1}P_j(a^{(N_2)}!*, b!*)$ meaning that a is a channel declared in network N_2 (whereas b is declared either in the current network or in one of its ancestors).
   A network maintains two lists of messages:
   - a list of current messages it must try to match; these messages were received previously;
   - messages it has handed over to its parent and which it remembers having sent; these messages cannot be matched by the current network.

   Some messages are handed over directly to the parent and are neither stored, nor remembered. (see below)
   A message should be forwarded to the parent in case:
   - it is non local; in this case the message is not remembered; or
   - it is mixed; in this case, it will be sent with some likelihood; this *likelihood* will change the bias towards local or global communications. If the message is not sent to the parent, it is stored in the current messages list. However, such a mixed message will only be sent to the parent if either
     - (a) it can match a mixed message sent earlier to the parent and which is in the remembered messages list, and (b) there are less than two remembered messages, or
     - no message is in the remembered list.

     In case the message is sent, the message is remembered.

   In the messages sent to the parent, the local channels are tagged with the *id*s of the network where they are local.
   When an alternative is sent, it is removed from the list of current alternatives. (followed by $\otimes$)
2. $\otimes$ (only done at the end of the other cases) *when the network is not receiving an input* (either from a child or from its parent), *it may attempt to find a match among its various alternatives*; first, it tries to find (randomly) a local or sublocal match;
   (a) If such a match is found, it is sent to the appropriate children.
   If there exists a match between two alternatives, both sublocal from the same network N, and if this match was not chosen, send them back.

(There will never me more than two from a given network.) If one was chosen, send the other back. (Each of the messages to which the alternatives belong can be mixed.) Remove them from the current messages. Then, remove the initial match from the current messages.

(b) If there is no sublocal match: if there is a current message with a sublocal channel, return it to the sender; if there is a superlocal match, send the two messages to the parent.

3. *the network receives an alternative back from its parent*: this corresponds to an alternative which was not used and is sent back by the parent; the current network puts the alternative back in the current list, and unmarks it as remembered (i.e., it is no longer in the remembered list). (followed by $\otimes$)
4. *the network receives a choice* (such as $P_1 : a!*$) *from its parent*. The corresponding item (for instance $P_1(a!*, b!*)$) is deleted from the remembered alternatives (if it is in the list), and then sent to the appropriate children. (followed by $\otimes$)

4 `Java` Implementation of the Scheduling Algorithm

The `Java` implementations of our guard-scheduling algorithms translate a VPL hierarchy of processes and networks into a corresponding hierarchy of "scheduling threads" in `Java`, so called because each such thread contains code for scheduling the threads it creates. A scheduling thread for a network defines children threads and calls them in parallel. A scheduling thread for a process executes the code corresponding to that process. An outline of the correspondence between VPL processes and networks and `Java` classes is given in figures 11 and 12.

A VPL process `p` has parameters, declares local types, local variables and (local) procedures, and has a body. The corresponding `Java` class declaration inherits from the `schedulingThread` class, as does also the network class. There are corresponding declarations for the local types and variables (see figure 15). The `Java` class declaration also has so-called "parameter variables" which are intended to store the values of the parameters of a class instantiation. Parameters are stored there when the constructor of the class is called. The constructor takes as parameters an integer `id` which identifies the process for its parent, the process parameters and a reference to its parent network which is of class `schedulingThread` too. The constructor of the `schedulingThread` class is called through `super` and the identifier, the number of children (here zero) and the parent identifier are passed to it. The constructor then initializes the parameter variables and sets the number of local channels to zero. In principle, the class declaration then contains methods for each procedure in the process, but in the current `Java` code, the procedures are actually expanded in the body of the process. The main code for the class is in the `run` method.

A VPL network `n` is similar to a VPL process except that it has local channels, network or process declarations, and its body is a list of networks or processes

in parallel. The corresponding `Java` code has variables for local channels. The constructor calls its superclass with k, the number of its children, which is an implicit parameter of the network declaration. The constructor also initializes an array of local channels that is used by the scheduling algorithm to check if a channel used in a communication is local to the network or not. After the constructor declaration, the classes corresponding to the subsystems are declared. The main method, `run`, declares an array of k threads, instantiates each of them, passing as parameters the identifier of the child, other parameters, and a lock which is the lock the child has to borrow according to the scheduling algorithm. Once the threads are declared, they are started.

The translation to `Java` is very straightforward and the `Java` output is very readable. An example of an excerpt corresponding to a `select` statement (from figure 13) is given in figure 14. This example uses the first scheduling algorithm. With the third scheduling algorithm, the only difference is the vanishing of the parent lock. The whole structure of the input is preserved, but `select` statements result in calls to the parent thread.

Scheduling threads use remote procedure calls for inter-thread communication. More precisely, in order to avoid a blocking of the sender, an intermediate thread is created with the information to pass to the scheduling thread. The intermediate thread calls the scheduling thread with an RPC. Scheduling threads also make use of a library implementing the basic VPL data types and channels. Messages communicated up the hierarchy of scheduling threads contain enough information to track the VPL-process threads from which the messages originated.

5 Verifying the Correctness of the Generated Code

As pointed out earlier, the generated code is very straightforward. The main concern of the verification is therefore not verifying the output, but verifying the implementation of the scheduling algorithm. To certify the correctness of the first scheduling algorithm, we use the Concurrency Factory itself. In particular, each time the code generator is invoked, a VPL abstraction of the generated code is also produced.

The Factory's local model checker may then be applied to the VPL code on a standard suite of correctness properties (see below) to enhance the user's confidence in the generated `Java` code. This strategy is reminiscent of the proof-carrying-code approach of Necula [Nec97].

The suite of modal mu-calculus formulas we assembled is targeted toward verifying the correctness of the `Java` implementation of our guard-scheduling algorithm. Let a represent an input command, an output command, or a `skip` statement. The suite includes the following formulas:

a can occur in the future: $\mu X.(\langle a\rangle tt \vee \langle -a\rangle X)$
a can always occur in the future: $\nu X.(\mu Y.(\langle a\rangle tt \vee \langle -a\rangle Y) \wedge [-]X)$
a will always occur in the future: $\mu X.([-a]X \wedge \langle -\rangle tt)$

Deadlock freedom: $\mu X.(\langle - \rangle tt \wedge [-]X)$
Livelock freedom: $\nu X.(\mu Y.([\tau]Y) \wedge [-]X)$

The syntax we use for formulas is from [BS90] and the reader is referred there for a detailed explanation of the logic's semantics. For the present paper's purposes, the English translation of the formulas should suffice.

We checked these formulas on a variety of test cases for the code generator, including a four-node version of the VPL specification of the Rether real-time ethernet protocol given in [DMN+97]. For many of the test cases, the Factory's model checker failed to terminate before running out of memory due to the rather complex nature of the scheduling code. We therefore simplified the VPL code generated by the Factory by applying (by hand) a simple partial-evaluation technique that lets networks know in advance the range of messages they will receive.

For the partially evaluated VPL code, the model checker terminated in all cases and found all the formulas in the test suite to be true, as desired, except for livelock freedom. The model checker revealed a livelock in one of our test cases in the form of a cycle of `skip` statements. The model checker's diagnostic facility further revealed that the livelock corresponded to a cycle of the `skip` alternatives used in the scheduling code to allow a network to proceed asynchronously with respect to its parent or children networks (see Section 3). Imposing fairness constraints, however, eradicates the livelock. We are currently examining how to automate the partial evaluation of the generated VPL abstraction of the `Java` code.

6 Testing

Besides model checking, another technique we employed to test the `Java` implementation of our guard-scheduling algorithm was to instrument generated `Java` code with debugging code and then run it. The resulting execution can be viewed as a high-level simulation of the scheduling algorithm, similar in effect to the way specifications are simulated in the Concurrency Factory itself. This approach was used to significant effect, helping us catch errors in the `Java` implementation that were abstracted away in the VPL representation of the algorithm. The `Java` simulation was also more robust than the Concurrency Factory on long simulations.

`Java`-based simulation proved particularly effective on the Rether-protocol case study [DMN+97]. Specifically, we implemented a graphical front-end to the generated `Java` code that ran as a separate thread. The generated code was instrumented to call a method of the graphical interface whenever a significant event was executed. The resulting graphical simulation proved highly effective in rendering an easy assimilated view of the protocol's behavior.

7 Conclusion

The idea of a hierarchical scheduling algorithm for the input/output guard scheduling problem is a natural one in the context of the Concurrency Factory, where system specifications are structured hierarchically in terms of networks and processes. Our hierarchical solutions can be viewed as a hybrid of previously proposed centralized and distributed algorithms. From the perspective of a parent node and its children, control is centralized at the parent; from the perspective of the overall hierarchy, control is distributed.

Generating concurrent `Java` code that correctly implements our hierarchical solution turned out to be tricky both with and without locks. The `Java` code produced by the Factory uses threads both for networks and processes. The scheduling algorithms were verified both by simulation (execution of the `Java` code) and with the Factory's local model checker (in the case of the first algorithm) on a number of test cases, including a four-node version of the Rether real-time ethernet protocol [DMN+97]. The model checker was instrumental in finding errors in a scaled down version of the scheduling algorithm.

8 Acknowledgments

Acknowledgments are due to Yuh-Jzer Joung for suggesting the bare bones of the lock-free algorithm.

References

[BS83] G. N. Buckley and A. Silberschatz. An effective implementation for the generalized input-output construct of CSP. *ACM Transactions on Programming Languages and Systems*, 5(2):223–235, 1983.

[BS90] J. Bradfield and C. Stirling. Verifying Temporal Properties of Processes. In J. C. M. Baeten and J. W. Klop, editors, *Proceedings of the First International Conference on Theories of Concurrency - Unification and Extension (CONCUR '90)*, volume 458 of *Lecture Notes in Computer Science*, pages 115–125. Springer-Verlag, 1990.

[CLSS96a] R. Cleaveland, P. M. Lewis, S. A. Smolka, and O. Sokolsky. The Concurrency Factory: A development environment for concurrent systems. In R. Alur and T. A. Henzinger, editors, *Proceedings of the 8th International Conference on Computer Aided Verification (CAV '96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 398–401, New Brunswick, New Jersey, 1996. Springer-Verlag.

[CLSS96b] R. Cleaveland, P. M. Lewis, S. A. Smolka, and O. Sokolsky. The Concurrency Factory software development environment. In T. Margaria and B. Steffen, editors, *Proceedings of the Second International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 391–395. Springer-Verlag, 1996.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design — A Foundation*. Addison-Wesley, 1988.

[DMN+97] X. Du, K. T. McDonnell, E. Nanos, Y. S. Ramakrishna, and S. A. Smolka. Software design, specification, and verification: Lessons learned from the Rether case study. In *Proceedings of the Sixth International Conference on Algebraic Methodology and Software Technology (AMAST'97)*, Sydney, Australia, December 1997. Springer-Verlag.

[Fla97] D. Flanagan. *Java in a Nutshell*. O'Reilly & Associates, Inc., Sebastopol, California, USA, 1997.

[FR80] N. Francez and M. Rodeh. A distributed abstract data type implemented by a probabilistic communication scheme. In *Proceedings of 21st Symposium on Foundations of Computer Science*, pages 373–379, 1980.

[Gar98] H. Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. Technical Report 3352, INRIA Rhône-Alpes, France, January 1998.

[GL98] S. J. Garland and N. A. Lynch. The IOA Language and Toolset: Support for Designing, Analyzing, and Building Distributed Systems. Technical Report MIT/LCS/TR-762, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1998.

[HLN+90] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. B. Trakhtenbrot. Statemate: A Working Environment for Development of Complex Reactive Systems. *IEEE Transaction on Software Engineering*, 16(4):403–414, April 1990.

[Ivy96] IvyTeam, Zurich, CH. *SystemSpecs: The Tool to Define, Analyse and Simulate Your System*, August 1996. `http://www.thenet.ch/tntech/sysspecs.html`.

[Nec97] G. C. Necula. Proof-Carrying Code. In *POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, January 1997.

[Ngu98] K. Nguyen. Scheduling algorithms used in Adagen for the Concurrency Factory, 1998. Department of Computer Science, SUNY at Stony Brook.

[OW97] S. Oaks and H. Wong. *Java Threads*. O'Reilly & Associates, Inc., Sebastopol, California, USA, 1997.

[RS97] Y. S. Ramakrishna and S. A. Smolka. Partial-order reduction in the weak modal mu-calculus. In A. Mazurkiewicz and J. Winkowski, editors, *Proceedings of the Eighth International Conference on Concurrency Theory (CONCUR '97)*, volume 1243 of *Lecture Notes in Computer Science*, Warsaw, Poland, July 1997. Springer-Verlag.

[Tiw97] P. K. Tiwari. VPL – Tool Support for Specification and Verification of Concurrent Systems. Master thesis, Department of Computer Science, North Carolina State University, 1997.

[TLK96] B. Thomsen, L. Leth, and T.-M. Kuo. A Facile Tutorial. In *Proceedings of the Seventh International Conference on Concurrency Theory (CONCUR '96)*, volume 1119 of *Lecture Notes in Computer Science*, pages 278–298. Springer-Verlag, 1996.

[VC95] C. Venkatramani and T. Chiueh. The design, implementation and evaluation of a software-based real-time ethernet protocol. In *Proceedings of ACM SIGCOMM '95*, pages 27–37, 1995.

```
initialize current and remembered messages to empty
while true
   select
      % // alternative 1
      begin
         receive a message from a child
         if it is non-local, forward it to the parent
         if it is mixed,
            send it (and remember it) to the parent with some likelihood, provided either
               (1)(a) it can match a mixed message sent earlier
                        to the parent and which is remembered, and
                     (b) there are less than two remembered messages, or
               (2) no message is remembered
         if it is not sent, store it in the current list
         try to find a match (see figure 10)
      end
      % // alternative 2
      begin
         receive an alternative back from the parent
         put it back in the current list
         unmark it as remembered
         try to find a match (see figure 10)
      end
      % // alternative 3
      begin
         receive a choice from the parent
         delete the corresponding item from the remembered alternatives (if any)
         send it to the appropriate children
         try to find a match (see figure 10)
      end
      % // alternative 4
      skip
   end select
end while
```

Fig. 9. Main network code (algorithm 3)

```
choose a match randomly among all (sub)local ones
if there is one
   begin
      send it to the appropriate children
      if there is another match between two alternatives,
         both sublocal from the same network N,
         and if this match is different from the one chosen
         send the alternatives back and remove them from the current messages
      else if there is another match between two alternatives,
         both sublocal from the same network N,
         and if only one of the alternatives was chosen,
         send the other back and remove it from the current messages
      end if
      remove the initial match from the current messages
   end
else
   begin
      if there is a current message with a sublocal channel,
         return it to the sender
      if there is a superlocal match, send the two messages to the parent
   end
end if
```

Fig. 10. Matching procedure (algorithm 3)

```
process p(⟨parameters⟩VPL)
   ⟨local types⟩VPL
   ⟨local variables⟩VPL
   ⟨procedures⟩VPL
begin
   ⟨body⟩VPL
end
```

$\Longrightarrow$

```
class p extends schedulingThread{
   ⟨parameter variables (variables for parameters)⟩java
   ⟨local types⟩java
   ⟨local variables⟩java
   // constructor
   p(int id, ⟨parameters⟩java ,
      schedulingThread parentScheduler) {
      super(id,0,parentScheduler);
      ⟨initialization of parameter variables⟩java
      ⟨initialization of local channels to empty⟩java
   }
   ⟨procedures⟩java
   public void run() {
      ⟨body⟩java
   }
}
```

Fig. 11. From a VPL process to a `Java` thread

```
network n(⟨parameters⟩VPL)                    class n extends schedulingThread{
    ⟨local types⟩VPL                             ⟨parameter variables (variables for parameters)⟩java
    ⟨local variables and channels⟩VPL            ⟨local types⟩java
    ⟨k networks or k processes⟩VPL      ⟹        ⟨local variables and channels⟩java
begin                                            // constructor
    ⟨networks or processes in parallel⟩VPL       n(int id, ⟨parameters⟩java ,
end                                                 schedulingThread parentScheduler) {
                                                    super(id,k, parentScheduler)
                                                    ⟨initialization of parameter variables⟩java
                                                    ⟨initialization of local channels array⟩java
                                                 }
                                                 ⟨networks or processes⟩java
                                                 public void run() {
                                                    schedulingThread st[]
                                                       = new schedulingThread[k]
                                                    st[0]=new ⟨network or process name⟩java(
                                                                0,⟨network or process parameters⟩java ,
                                                                this);
                                                    ...
                                                    st[k − 1]=new ...(k − 1,...);
                                                    st[0].start();
                                                    ...
                                                    st[k − 1].start();
                                                 }
                                              }
```

Fig. 12. From a VPL network to a Java thread

```
process p1(a: synch,b: synch)
  begin
    while true do
      select
        a!*
        %
        b?*
      end
    end
  end; {end p1}
```

Fig. 13. Excerpt of the VPL code

```
class p1 extends SchedulingThread {
  // parameters
  private SyncChannel a;
  private SyncChannel b;

  // local variables

  // constructor
  p1(int id,
     SyncChannel a, SyncChannel b,
     SchedulingThread parentScheduler, Object parentLock) {
    super(id, // id
      0, // number of children
      parentScheduler, parentLock,
      "p1");
    this.a = a;
    this.b = b;
    // no local channels
    setNumberOfLocalChannels(0);
    }
  public void run() {
    Message[] messages;
    while (1==1)
      {
      messages = new Message[2];
      messages[0] = new Message(a,true,"a!*");
      messages[1] = new Message(b,false,"b?*");
      reportToScheduler("p1",messages, 2);
      if (getDecision().isAction1(a,true,"a!*"))
        a.syncSend();
      else if (getDecision().isAction1(b,false,"b?*"))
        b.syncReceive();
      }
  } // end of run method
} // end of class p1
```

Fig. 14. Excerpt of the code generated for the `select` statement in figure 13 (algorithm 1).

```
type tok_var : record
  node: boolean;
  m : count_type;
end
```

```
class tok_var {
  Bool node = new Bool();
  count_type m = new count_type();
  // constructor
  tok_var() { }
  // copy constructor
  tok_var(tok_var _tok_var) {
    node = new Bool(_tok_var.node);
    m = new count_type(_tok_var.m);
  }
  // set
  public void set(tok_var _tok_var) {
    node.set(_tok_var.node);
    m.set(_tok_var.m);
  }
}
```

Fig. 15. Example of type correspondence between VPL and Java